

VISUAL AND SCENE GRAPH INTERFACES

CROSS-REFERENCE TO RELATED APPLICATIONS

5 The present invention is a continuation-in-part of United States Patent Application No. 10/402,268 filed March 27, 2003.

FIELD OF THE INVENTION

10 The invention relates generally to computer systems, and more particularly to the processing of graphical and other video information for display on computer systems.

BACKGROUND OF THE INVENTION

15 The limits of the traditional immediate mode model of accessing graphics on computer systems are being reached, in part because memory and bus speeds have not kept up with the advancements in main processors and/or graphics processors. In general, the current (e.g., WM_PAINT) model for preparing a frame requires too much data processing to keep up with the hardware refresh rate when complex graphics effects are
20 desired. As a result, when complex graphics effects are attempted with conventional graphics models, instead of completing the changes that result in the perceived visual effects in time for the next frame, the changes may be added

over different frames, causing results that are visually and noticeably undesirable.

A new model for controlling graphics output is described in United States Patent Application Serial Nos. 10/184,795,
5 10/184,796, 10/185,775, 10/401,717, 10/402,322 and 10/402,268, assigned to the assignee of the present invention and hereby incorporated by reference. This new model provides a number of significant improvements in graphics processing technology. For example, U.S. Serial No. 10/184,795 is generally directed
10 towards a multiple-level graphics processing system and method, in which a higher-level component (e.g., of an operating system) performs computationally intensive aspects of building a scene graph, updating animation parameters and traversing the scene graph's data structures, at a relatively low operating
15 rate, in order to pass simplified data structures and/or graphics commands to a low-level component. Because the high-level processing greatly simplifies the data, the low-level component can operate at a faster rate, (relative to the high-level component), such as a rate that corresponds to the frame
20 refresh rate of the graphics subsystem, to process the data into constant output data for the graphics subsystem. When animation is used, instead of having to redraw an entire scene with changes, the low-level processing may interpolate parameter intervals as necessary to obtain instantaneous values

that when rendered provide a slightly changed scene for each frame, providing smooth animation.

U.S. Serial No. 10/184,796 describes a parameterized scene graph that provides mutable (animated) values and parameterized graph containers such that program code that wants to draw
5 graphics (e.g., an application program or operating system component) can selectively change certain aspects of the scene graph description, while leaving other aspects intact. The program code can also reuse already-built portions of the scene
10 graph, with possibly different parameters. As can be appreciated, the ability to easily change the appearance of displayed items via parameterization and/or the reuse of existing parts of a scene graph provide substantial gains in overall graphics processing efficiency.

U.S. Serial No. 10/185,775 generally describes a caching
15 data structure and related mechanisms for storing visual information via objects and data in a scene graph. The data structure is generally associated with mechanisms that intelligently control how the visual information therein is
20 populated and used. For example, unless specifically requested by the application program, most of the information stored in the data structure has no external reference to it, which enables this information to be optimized or otherwise processed. As can be appreciated, this provides efficiency and
25 conservation of resources, e.g., the data in the cache data

structure can be processed into a different format that is more compact and/or reduces the need for subsequent, repeated processing, such as a bitmap or other post-processing result.

While the above improvements provide substantial benefits
5 in graphics processing technology, there still needs to be a way for programs to effectively use this improved graphics model and its other related improvements in a straightforward manner. What is needed is a comprehensive yet straightforward model for programs to take advantage of the many features and
10 graphics processing capabilities provided by the improved graphics model and thereby output complex graphics and audiovisual data in an efficient manner.

SUMMARY OF THE INVENTION

15 Briefly, the present invention provides an object model, and an application programming interface (API) for accessing that object model in a manner that allows program code developers to consistently interface with a scene graph data structure to produce graphics. A base object in the model and
20 API set is a visual, which represents a virtual surface to the user; the scene graph is built of visual objects. Such Visuals include container visual objects, retained visual objects, drawing visual objects and other visual objects. Visuals themselves can hold onto resource objects, such as clip
25 objects, transform objects and so forth, and some type of

Visuals (e.g., DrawingVisual, RetainedVisual) can hold on to drawing instruction lists that may reference resource objects, such as images, brushes and/or gradients.

Most resource objects in the scene graph are immutable
5 once created, that is, once they are created they cannot be changed. For those objects that a developer wants to easily change, mutability is provided by a changeables pattern and implementation, as described in copending United States Patent Application entitled "Changeable Class and Pattern to Provide
10 Selective Mutability in Computer Programming Environments" filed concurrently herewith, assigned to the assignee of the present invention and herein incorporated by reference.

Via the application programming interfaces, program code writes drawing primitives such as geometry data, image data,
15 animation data and other data to the visuals. For example, program code writes drawing primitives such as draw line instructions, draw geometry instructions, draw bitmap instructions and so forth into the visuals. Those drawing instructions are often combined with complex data like geometry
20 data that describes how a path is drawn, and they also may reference resources like bitmaps, videos, and so forth.

The code can also specify clipping, opacity and other properties on visuals, and methods for pushing and popping transform, opacity and hit test identification are provided.
25 In addition, the visual may participate in hit testing. The

program code also interfaces with the visuals to add child
visuals, and thereby build up a hierarchical scene graph. A
visual manager processes (e.g., traverses or transmits) the
scene graph to provide rich graphics data to lower-level
5 graphics components.

Container visuals provide for a collection of children
visuals and in one implementation, are the only visuals that
can define hierarchy. The collection of children on a
container visual allows for arbitrary insertion, removal and
10 reordering of children visuals.

Drawing visuals are opened with an open call that returns
a drawing context (e.g., a reference to a drawing context
object) to the caller. In general, a drawing context is a
temporary helper object that is used to populate a visual. The
15 program code then uses the drawing context to add drawing
primitives to the visual. The open call may clear the contents
(children) of a visual, or an append call may be used to open a
visual for appending to that current visual. In addition to
receiving static values as drawing parameters, drawing contexts
20 can be filled with animation objects.

A retained visual operates in a similar manner to a
drawing visual, except that its drawing context is filled when
the system requests that it be filled, instead of when the
program code wants to fill it. For example, if a particular
25 Visual's content will be needed in rendering a scene, the

system will call `IRetainedVisual.Render` to fill the content of the Visual, replacing any content already in memory.

Thus, different types of primitives may be drawn into a visual using a drawing context, including geometry, image data
5 and video data. Geometry is a type of class that defines a vector graphics skeleton without stroke or fill, e.g., a rectangle. Each geometry object corresponds to a simple shape (`LineGeometry`, `EllipseGeometry`, `RectangleGeometry`), a complex single shape (`PathGeometry`), or a list of such shapes
10 (`GeometryList`) with a combine operation specified (e.g., union, intersection and so forth.) These objects form a class hierarchy. There are also shortcuts for drawing frequently used types of geometry, such as a `DrawRectangle` method.

When geometry is drawn, a brush or pen may be specified.
15 A brush object defines how to graphically fill a plane, and there is a class hierarchy of brush objects. A pen also has a brush specified on it describing how to fill the stroked area. A special type of brush object (the `VisualBrush`) can reference a visual to define how that brush is to be drawn. A drawing
20 brush makes it possible to fill a shape or control with combinations of other shapes and brushes.

Other benefits and advantages will become apparent from the following detailed description when taken in conjunction with the drawings, in which:

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 is a block diagram representing an exemplary computer system into which the present invention may be incorporated;

5 FIG. 2 is a block diagram generally representing a graphics layer architecture into which the present invention may be incorporated;

FIG. 3 is a representation of a scene graph of visuals and associated components for processing the scene graph such as by
10 traversing the scene graph to provide graphics commands and other data in accordance with an aspect of the present invention;

FIG. 4 is a representation of a scene graph of validation visuals, drawing visuals and associated Instruction Lists
15 constructed in accordance with an aspect of the present invention;

FIG. 5 is a representation of a visual class, of an object model, in accordance with an aspect of the present invention;

FIG. 6 is a representation of various other objects of the
20 object model, in accordance with an aspect of the present invention;

FIG. 7 is a representation of a transform class hierarchy, in accordance with an aspect of the present invention;

FIGS. 8 and 9 are representations of transformations of a
25 visual's data in a geometry scale and a non-uniform scale,

respectively, in accordance with an aspect of the present invention;

FIG. 10 is a representation of geometry classes of the object model, in accordance with an aspect of the present
5 invention;

FIG. 11 is a representation of a PathGeometry structure, in accordance with an aspect of the present invention;

FIG. 12 is a representation of a scene graph of visuals and Instruction Lists showing example graphics produced by the
10 primitives, in accordance with an aspect of the present invention;

FIG. 13 is a representation of brush classes of the object model, in accordance with an aspect of the present invention;

FIGS. 14 and 15 are representations of rendered graphics
15 resulting from data in a linear gradient brush object, in accordance with an aspect of the present invention;

FIG. 16 is a representation of rendered graphics resulting from data in a radial gradient brush object, in accordance with an aspect of the present invention;

20 FIG. 17 is a representation of a rendered nine grid brush object in accordance with an aspect of the present invention.

FIG. 18 is a representation of rendered graphics resulting from having various stretch values, in accordance with an aspect of the present invention;

FIG. 19 is a representation of rendered graphics resulting from having various tile values, in accordance with an aspect of the present invention;

FIG. 20 is a representation of a grid and transformed
5 grid, resulting from data in a visual brush object, in accordance with an aspect of the present invention;

FIG. 21 is a representation of the grid and transformed grid, with rendered graphics therein drawn from a visual, in accordance with an aspect of the present invention.

10

DETAILED DESCRIPTION

EXEMPLARY OPERATING ENVIRONMENT

FIGURE 1 illustrates an example of a suitable computing system environment 100 on which the invention may be
15 implemented. The computing system environment 100 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment 100 be interpreted as having any dependency or
20 requirement relating to any one or combination of components illustrated in the exemplary operating environment 100.

The invention is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems,
25 environments, and/or configurations that may be suitable for

use with the invention include, but are not limited to, personal computers, server computers, hand-held or laptop devices, tablet devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable
5 consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

The invention may be described in the general context of computer-executable instructions, such as program modules,
10 being executed by a computer. Generally, program modules include routines, programs, objects, components, data structures, and so forth, which perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where
15 tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

20 With reference to FIG. 1, an exemplary system for implementing the invention includes a general purpose computing device in the form of a computer 110. Components of the computer 110 may include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121 that
25 couples various system components including the system memory

to the processing unit 120. The system bus 121 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not
5 limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, Accelerated Graphics Port (AGP) bus, and Peripheral Component Interconnect (PCI) bus also known
10 as Mezzanine bus.

The computer 110 typically includes a variety of computer-readable media. Computer-readable media can be any available media that can be accessed by the computer 110 and includes both volatile and nonvolatile media, and removable and non-
15 removable media. By way of example, and not limitation, computer-readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of
20 information such as computer-readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes,
25 magnetic tape, magnetic disk storage or other magnetic storage

devices, or any other medium which can be used to store the desired information and which can be accessed by the computer 110. Communication media typically embodies computer-readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of the any of the above should also be included within the scope of computer-readable media.

15 The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1 illustrates operating

system 134, application programs 135, other program modules 136 and program data 137.

The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1 illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156 such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through a non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150.

The drives and their associated computer storage media, discussed above and illustrated in FIG. 1, provide storage of computer-readable instructions, data structures, program modules and other data for the computer 110. In FIG. 1, for example, hard disk drive 141 is illustrated as storing

operating system 144, application programs 145, other program modules 146 and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers herein to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a tablet (electronic digitizer) 164, a microphone 163, a keyboard 162 and pointing device 161, commonly referred to as mouse, trackball or touch pad. Other input devices (not shown) may include a joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190. The monitor 191 may also be integrated with a touch-screen panel 193 or the like that can input digitized input such as handwriting into the computer system 110 via an interface, such as a touch-screen interface 192. Note that the monitor and/or touch screen panel can be physically coupled to a housing in which

the computing device 110 is incorporated, such as in a tablet-type personal computer, wherein the touch screen panel 193 essentially serves as the tablet 164. In addition, computers such as the computing device 110 may also include other
5 peripheral output devices such as speakers 195 and printer 196, which may be connected through an output peripheral interface 194 or the like.

The computer 110 may operate in a networked environment using logical connections to one or more remote computers, such
10 as a remote computer 180. The remote computer 180 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer 110, although only a memory storage device 181 has
15 been illustrated in FIG. 1. The logical connections depicted in FIG. 1 include a local area network (LAN) 171 and a wide area network (WAN) 173, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

20 When used in a LAN networking environment, the computer 110 is connected to the LAN 171 through a network interface or adapter 170. When used in a WAN networking environment, the computer 110 typically includes a modem 172 or other means for establishing communications over the WAN 173, such as the
25 Internet. The modem 172, which may be internal or external,

may be connected to the system bus 121 via the user input interface 160 or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 110, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, FIG. 1 illustrates remote application programs 185 as residing on memory device 181. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

10

INTERFACES TO SCENE GRAPH DATA STRUCTURES

One aspect of the present invention is generally directed to allowing program code, such as an application or operating system component, to communicate drawing instructions and other information (e.g., image bitmaps) to graphics components in order to render graphical output on the system display. To this end, the present invention provides a number of defined functions and methods, e.g., in the form of application programming interfaces (APIs) to an object model, that enable programs to populate a scene graph with data structures, drawing primitives (commands), and other graphics-related data. When processed, the scene graph results in graphics being displayed on the screen.

FIG. 2 represents a general, layered architecture 200 into which the present invention may be implemented. As represented

in FIG. 2, program code 202 (e.g., an application program or operating system component or the like) may be developed to output graphics data in one or more various ways, including via imaging 204, via vector graphic elements 206, and/or via
5 function / method calls placed directly to a visual application programming interface (API) layer 212, in accordance with an aspect of the present invention. In general, imaging 204 provides the program code 202 with a mechanism for loading, editing and saving images, e.g., bitmaps. As described below,
10 these images may be used by other parts of the system, and there is also a way to use the primitive drawing code to draw to an image directly. Vector graphics elements 206 provide another way to draw graphics, consistent with the rest of the object model (described below). Vector graphic elements 206
15 may be created via a markup language, which an element / property system 208 and layout system 210 interprets to make appropriate calls to the visual API layer 212. Vector graphic elements 206, along with the element / property system 208 and layout system 210, are described in United States patent
20 application Serial No. 10/401,717.

In one implementation, the graphics layer architecture 200 includes a high-level composition and animation engine 214, which includes or is otherwise associated with a caching data structure 216. The caching data structure 216 contains a scene
25 graph comprising hierarchically-arranged objects that are

managed according to a defined object model, as described below. In general, the visual API layer 212 provides the program code 202 (and the layout system 210) with an interface to the caching data structure 216, including the ability to
5 create objects, open and close objects to provide data to them, and so forth. In other words, the high-level composition and animation engine 214 exposes a unified media API layer 212 by which developers may express intentions about graphics and media to display graphics information, and provide an
10 underlying platform with enough information such that the platform can optimize the use of the hardware for the program code. For example, the underlying platform will be responsible for caching, resource negotiation and media integration.

In one implementation, the high-level composition and
15 animation engine 214 passes an instruction stream and possibly other data (e.g., pointers to bitmaps) to a fast, low-level compositing and animation engine 218. As used herein, the terms "high-level" and "low-level" are similar to those used in other computing scenarios, wherein in general, the lower a
20 software component is relative to higher components, the closer that component is to the hardware. Thus, for example, graphics information sent from the high-level composition and animation engine 214 may be received at the low-level compositing and animation engine 218, where the information is used to send

graphics data to the graphics subsystem including the hardware
222.

The high-level composition and animation engine 214 in
conjunction with the program code 202 builds a scene graph to
5 represent a graphics scene provided by the program code 202.
For example, each item to be drawn may be loaded with drawing
instructions, which the system can cache in the scene graph
data structure 216. As will be described below, there are a
number of various ways to specify this data structure 216, and
10 what is drawn. Further, the high-level composition and
animation engine 214 integrates with timing and animation
systems 220 to provide declarative (or other) animation control
(e.g., animation intervals) and timing control. Note that the
animation system allows animate values to be passed essentially
15 anywhere in the system, including, for example, at the element
property level 208, inside of the visual API layer 212, and in
any of the other resources. The timing system is exposed at
the element and visual levels.

The low-level compositing and animation engine 218
20 manages the composing, animating and rendering of the scene,
which is then provided to the graphics subsystem 222. The low-
level engine 218 composes the renderings for the scenes of
multiple applications, and with rendering components,
implements the actual rendering of graphics to the screen.
25 Note, however, that at times it may be necessary and/or

advantageous for some of the rendering to happen at higher levels. For example, while the lower layers service requests from multiple applications, the higher layers are instantiated on a per-application basis, whereby is possible via the imaging mechanisms 204 to perform time-consuming or application-specific rendering at higher levels, and pass references to a bitmap to the lower layers.

In accordance with an aspect of the present invention, a Visual application programming interface (API) provides a set of objects that define the visual tree in accordance with an aspect of the present invention. The visual tree represents a data structure that can be rendered by the graphics system to a medium (a screen, printer or surface). When rendered, the data in the visual tree is the "scene" a viewer sees. Visual objects (or simply Visuals) contain and manage the other graphical objects that make up a drawn scene like geometries, primitives, brushes, color gradients, and animations.

Although the present invention also provides access to drawing and rendering services at a higher level of abstraction using a more familiar object and property system, and provides vector graphics objects at the markup level through markup language (code-named "XAML"), the Visual API will be of most interest to developers who want greater control over drawing functionality than they can easily achieve using the property system or markup.

FIGS. 3 and 4 show example scene graphs 300 and 400, respectively, including the base object referred to as a Visual. Visual Objects, or simply visuals, are containers for graphical content such as lines, text, and images. As
5 represented in the object inheritance of the visual classes in FIG. 5, there are several different visual objects, including ContainerVisual, which is a Visual that does not directly contain graphical content, but contains child Drawing Visual objects. Child DrawingVisual objects are added to a
10 ContainerVisual rather than to other DrawingVisual objects. This allows the developer to make changes and set properties on individual visual objects without recreating and then re-rendering the entire drawing context, while also allowing access to clipping and transform properties on the container
15 object. ContainerVisual objects can be nested.

A DrawingVisual is a Visual that can contain graphical content. This Visual exposes a number of drawing methods. The child objects of a DrawingVisual are organized in a zero-based, z-order space. A RetainedVisual is A Visual that introduces a
20 "retained instruction stream" that can be used for drawing. In simpler terms, the RetainedVisual allows the developer to retain the visual's content and redraw it only when necessary. It is possible to use the RetainedVisual imperatively, like a DrawingVisual, by calling RenderOpen and using the returned
25 DrawingContext for drawing. The RetainedVisual provides

validation callback functionality and an InvalidateVisual method. To use validation functionality, the user implements the IRetainedRender interface on the RetainedVisual or a class that derives from it.

5 Returning to FIG. 5, yet another visual is an HwndVisual 505, which is a Visual used to host a legacy Microsoft® Win32® control or window as a child visual inside a visual scene of a scene graph. More particularly, legacy programs will still operate via the WM_PAINT method (or the like) that draws to a
10 child HWND (or the like) based on prior graphics technology. To support such programs in the new graphics processing model, the HwndVisual allows the HWND to be contained in a scene graph and moved as the parent visual is repositioned. Other types of visuals are also feasible, such as three-dimensional (3D)
15 visuals which enable a connection between two-dimensional and three dimensional worlds, e.g., a camera-like view is possible via a two-dimensional visual having a view into a three-dimensional world.

As shown in FIG. 3, a VisualManager 304 comprises an
20 object that connects a visual tree to a medium. The VisualManager establishes a retained connection between the visual data structure (the root visual 302) and the target to which that data is rendered, offering the possibility of tracking differences between the two. The VisualManager 304
25 receives window messages and provides methods for transforming

a point in drawing coordinates to device coordinates and vice versa.

A typical application might draw graphics by defining a layout in "XAML" as described in the aforementioned United States patent application Serial No. 10/401,717, and also by specifying some drawing operations in C#. Developers may create Shape elements, or draw geometries using the Geometry classes with primitives. In the following scenario, the code demonstrates drawing an ellipse in the Visual that underlies the Canvas:

```
private void CreateAndShowMainWindow ()
{
    mainWindow = new MSAvalon.Windows.Window ();

    Canvas myCanvas = new Canvas();
    mainWindow.Children.Add(myCanvas);

    Ellipse e1 = new Ellipse();
    e1.Fill = Brushes.Blue;
    e1.Stroke = Brushes.Black;
    e1.StrokeThickness = new Length(10);
    e1.CenterX = new Length(100);
    e1.CenterY = new Length(75);
    e1.RadiusX = new Length(50);
    e1.RadiusY = new Length(50);

    myCanvas.Children.Add(e1);
    mainWindow.Show();
}
```

Using the Visual API, developers can instead draw directly into the Visual (that would otherwise be accessed via by the layout element).

To render the content of a `DrawingVisual` object, an application typically calls the `RenderOpen` method on the `DrawingVisual`. `RenderOpen` returns a `DrawingContext` with which the application can perform drawing operations. To clear the
5 `Visual`'s contents, the application calls `Close` on the `DrawingContext`. After the application calls `Close`, the `DrawingContext` can no longer be used.

The following code draws an ellipse (the same ellipse as in the previous example) into a `DrawingVisual`, using a `Geometry`
10 object rather than the `Ellipse` shape. The example creates a `DrawingVisual`, gets the `DrawingVisual`'s `DrawingContext`, and calls the `DrawingContext`'s `DrawGeometry` method to draw the ellipse. Note that you must add the `Visual` to the visual tree of the top-level object, which in this case is the window.

```
mainWindow = new MSAvalon.Windows.Window();

    mainWindow.Show();

    DrawingVisual myDrawingVisual = new
DrawingVisual();
    DrawingContext myDrawingContext =
myDrawingVisual.RenderOpen();

    SolidColorBrush mySolidColorBrush = new
SolidColorBrush();
    mySolidColorBrush.Color = Colors.Blue;
    Pen myPen = new Pen(Brushes.Black, 10);

    EllipseGeometry aGeometry = new EllipseGeometry(new
Point(100,75), 50, 50);
    myDrawingContext.DrawGeometry(mySolidColorBrush,
myPen, aGeometry);
    myDrawingContext.Close();
```

```
((IVisual)mainWindow).Children.Add(myDrawingVisual)
```

The following example further builds on the previous example by adding similar ellipses to a ContainerVisual; note that this example is verbose for clarity). Using ContainerVisual can help organize scene objects and allow the developer to segregate Visual objects on which to perform hit-testing or validation (RetainedVisual objects) from ordinary drawn content, and minimize unnecessary redrawing of content.

```
        mainWindow = new MS Avalon.Windows.Window();

        mainWindow.Show();

//Create some Visuals
        ContainerVisual myContainer = new ContainerVisual();

        DrawingVisual myDrawingVisual = new DrawingVisual();
        DrawingVisual myDrawingVisual_1 = new DrawingVisual();
        DrawingVisual myDrawingVisual_2 = new DrawingVisual();

//Perform some drawing
        DrawingContext myDrawingContext =
myDrawingVisual.RenderOpen();
        SolidColorBrush mySolidColorBrush = new SolidColorBrush();
mySolidColorBrush.Color = Colors.Violet;
        Pen myPen = new Pen(Brushes.Black, 10);
        EllipseGeometry aGeometry = new EllipseGeometry(new
Point(100,75), 50, 50);
        myDrawingContext.DrawGeometry(mySolidColorBrush, myPen,
aGeometry);
        myDrawingContext.Close();

        DrawingContext myDrawingContext_1 =
myDrawingVisual_1.RenderOpen();
        mySolidColorBrush.Color = Colors.Red;
        Pen myPen1 = new Pen(Brushes.Orange, 10);
        EllipseGeometry aGeometry1 = new EllipseGeometry(new
Point(100,175), 50, 50);
        myDrawingContext_1.DrawGeometry(mySolidColorBrush, myPen1,
aGeometry1);
        myDrawingContext_1.Close();

        DrawingContext myDrawingContext_2 =
myDrawingVisual_2.RenderOpen();
        mySolidColorBrush.Color = Colors.Yellow;
        Pen myPen2 = new Pen(Brushes.Blue, 10);
```

```

        EllipseGeometry aGeometry2 = new EllipseGeometry(new
Point(100,275), 50, 50);
        myDrawingContext_2.DrawGeometry(mySolidColorBrush, myPen2,
aGeometry2);
        myDrawingContext_2.Close();

//Add DrawingVisuals to the ContainerVisual's VisualCollection
        myContainer.Children.Add(myDrawingVisual);
        myContainer.Children.Add(myDrawingVisual_1);
        myContainer.Children.Add(myDrawingVisual_2);

//Add the ContainerVisual to the window
        ((IVisual)mainWindow).Children.Add(myContainer);

```

A RetainedVisual is similar to a DrawingVisual, but allows for selective redrawing of visual content. As its name suggests, the RetainedVisual can retain content for multiple appearances on the medium. It also provides callback and validation functionality. This functionality can help with rendering performance by offering the developer greater control over re-rendering of content.

At a basic level, the user can create and use a RetainedVisual much like a DrawingVisual; that is, the user can call RenderOpen and get a DrawingContext. Alternatively, the user can implement the IRetainedRender interface on a RetainedVisual. By doing so, users ensure that the graphics system will use the value set in the RenderBounds property as the bounds for the content to be rendered at the IRetainedVisual.Render call.

When rendering the scene, the graphics system will examine any child Visual. If the value of the RenderBounds property indicates that a particular Visual's content will be needed in rendering a scene, the system will call IRetainedVisual.Render

to fill the content of the Visual, replacing any content already in memory. The application can also call InvalidateVisual directly to flush content from a Visual. If the application has not implemented IRetainedRender on the RetainedVisual, any call to InvalidateVisual will throw an exception.

The following code instantiates a class that implements IRetainedRender on a RetainedVisual and draws into it.

```
public class Rectangle : RetainedVisual, IRetainedRender
{
    public Rectangle(Color color, Rect rect) :
base()
    {
        m_color = color;
        m_rect = rect;
        RenderBounds = rect;
    }
    public void SetColor(Color color)
    {
        m_color = color;
        InvalidateVisual();
    }
    public void Render(DrawingContext ctx)
    {
        ctx.DrawRectangle(
            new SolidColorBrush(m_color),
            null,
            m_rect);
    }
}
```

10

The Visual API, like the rest of the graphics system of the present invention, is a managed API and makes use of typical features of managed code, including strong typing and

garbage collection. It also takes advantage of the hardware acceleration capability for rendering. To accommodate developers working with existing unmanaged applications, the Visual API provides limited interoperability between the present graphics system and Microsoft Windows® Graphics Device Interface (GDI)-based rendering services.

This interoperability allows developers to host GDI-based windows in Visual-aware applications using the Hwnd Visual object, write controls and theming that are based on the present invention's drawing and rendering, but still work in legacy GDI applications, and Modify GDI HWND-based applications to take advantage of the new rendering features, including hardware acceleration and the color model.

The HwndVisual enables hosting of Win32 content in a Visual-aware application. As represented in FIG. 5, HwndVisual inherits from ContainerVisual. Note that it is not possible to mix GDI and the new drawing models in the same HwndVisual. Instead, this visual might be more useful for legacy controls of limited scope. The following example demonstrates creating a control in an HwndVisual and adding it to the visual tree.

```
//Import Win32 resources and define variables for a control.  
.  
.  
.  
//Create the control.  
hwndControl = CreateWindowEx(  
0,  
WC_TREEVIEW,  
"",  
WS_CHILD | WS_VISIBLE | TVS_HASLINES | TVS_LINESATROOT |
```

```

TVS_HASBUTTONS,
x,
y,
cx,
cy,
hwndParent,
IntPtr.Zero,
IntPtr.Zero,
0);

//Create an HwndVisual for the control and add it to a
previously-defined
//collection.
s_visual1 = HwndVisual.GetHwndVisual(hwndControl);
s_visual1.Size = new Size (150, 150);
s_visual1.IsHwndDpiAware = false;

s_visual0.Children.Add(s_visual1);

```

As with other objects, you can apply transforms and other property changes to the control once hosted in a Visual.

```

TransformCollection t = new TransformCollection();
t.AddScale(1.4, 1.4);
t.AddTranslate(130, 80);
s_visual0.Children.SetTransform(s_visual1, t);

```

5

As represented in FIG. 3, a top-level (or root) Visual 302 is connected to a Visual manager object 304, which also has a relationship (e.g., via a handle) with a window (HWND) 306 or similar unit in which graphic data is output for the program code. The VisualManager 304 manages the drawing of the top-level Visual (and any children of that Visual) to that window 306. FIG. 6 shows the VisualManager as one of a set of objects

600 in the object model of the graphics system described herein.

To draw, the Visual manager 304 processes (e.g., traverses or transmits) the scene graph as scheduled by a dispatcher 308, and provides graphics instructions and other data to the low level component 218 (FIG. 2) for its corresponding window 306, such as generally described in the aforementioned U.S. Patent Applications. The scene graph processing will ordinarily be scheduled by the dispatcher 308 at a rate that is relatively slower than the refresh rate of the lower-level component 218 and/or graphics subsystem 222.

FIG. 3 shows a number of child Visuals 310-314 arranged hierarchically below the top-level (root) Visual 302, some of which are represented as having been populated via drawing contexts 316, 317 (shown as dashed boxes to represent their temporary nature) with associated instruction lists 318 and 319, respectively, e.g., containing Instruction Lists and other Visuals. The Visuals may also contain other property information. In general, most access on the base visual class comes via an IVisual interface, and visual derives from DependencyObject, as represented in FIG. 5. Among other drawing primitives, the instruction list may include a reference to an ImageData. That ImageData can then be changed/updated directly by getting a drawing context off of

it, or by having a SurfaceVisualRenderer (alternatively named ImageDataVisualRenderer).

Visuals offer services by providing clip, opacity and possibly other properties that can be set, and/or read via a get method. In addition, the visual has flags controlling how it participates in hit testing. A Show property is used to show/hide the visual, e.g., when false the visual is invisible, otherwise the visual is visible. Furthermore, these objects (whether Visuals at the Visual API layer or elements at the element layer) exist in a hierarchy. A coordinate system is inherited down through this hierarchy. In this way, a parent can push a coordinate transform that modifies the rendering pass and gets applied to that parent's children.

The transform for a visual is on the connection to that visual. In other words, it is set via the [Get|Set]ChildTransform on the parent's IVisual interface.

Note that the coordinate transforms may be applied in a uniform way to everything, as if it were in a bitmap. Note that this does not mean that transformations always apply to bitmaps, but that what gets rendered is affected by transforms equally. By way of example, if the user draws a circle with a round pen that is one inch wide and then applies a scale in the X direction of two to that circle, the pen will be two inches wide at the left and right and only one inch wide at the top and bottom. This is sometimes referred to as a compositing or

bitmap transform (as opposed to a skeleton or geometry scale that affects the geometry only). FIG. 8 is a representation of scaling transformation, with a non-transformed image 800 appearing on the left and a transformed image 802 with a non-uniform scale appearing on the right. FIG. 9 is a representation of scaling transformation, with the non-transformed image 800 appearing on the left and a transformed image 904 with geometry scaling appearing on the right.

With respect to coordinate transformation of a visual, TransformToDescendant transforms a point from the reference visual to a descendant visual. The point is transformed from the post-transformation coordinate space of the reference visual to the post-transformation coordinate space of the descendant visual. TransformFromDescendant transforms a point from the descendant visual up the parent chain to the reference visual. The point is transformed from post-transformation coordinate space of the descendant visual to post-transformation coordinate space of the reference visual. A user may get a Matrix to and from a descendant and from and to any arbitrary visual. Two properties are available that may be used to determine the bounding box of the content of the Visual, namely DescendantBounds, which is the bounding box of the descendants, and ContentBounds which is the bounds of the content. Applying a Union to these provides the total bounds.

The clip property sets (and gets) the clipping region of a visual. Any Geometry (the geometry class is shown in FIG. 10) can be used as a clipping region, and the clipping region is applied in Post-Transformation coordinate space. In one
5 implementation, a default setting for the clipping region is null, i.e., no clipping, which can be thought of as an infinite big clipping rectangle from $(-\infty, -\infty)$ to $(+\infty, +\infty)$.

The Opacity property gets/sets the opacity value of the visual, such that the content of the visual is blended on the
10 drawing surface based on the opacity value and the selected blending mode. The BlendMode property can be used to set (or get) the blending mode that is used. For example, an opacity (alpha) value may be set between 0.0 and 1.0, with linear alpha blending set as the mode, e.g., $\text{Color} = \text{alpha} * \text{foreground}$
15 $\text{color} + (1.0 - \text{alpha}) * \text{background color}$. Other services, such as special effects properties, may be included in a visual, e.g., blur, monochrome, and so on.

The various services (including transform, opacity, and clip) can be pushed and popped on a drawing context, and push/
20 pop operations can be nested, as long as there is an appropriate pop call for each push call.

The PushTransform method pushes a transformation. Subsequent drawing operations are executed with respect to the pushed transformation. The pop call pops the transformation
25 pushed by the matching PushTransform call:

```
void PushTransform(Transform transform);  
void PushTransform(Matrix matrix);  
void Pop();.
```

5

Similarly, the PushOpacity method pushes an opacity value. Subsequent drawing operations are rendered on a temporary surface with the specified opacity value and then composite into the scene. Pop() pops the opacity pushed by the matching

10 PushOpacity call:

```
void PushOpacity(float opacity);  
void PushOpacity(FloatAnimation opacity);  
void Pop();.
```

15

The PushClip method pushes a clipping geometry. Subsequent drawing operations are clipped to the geometry. The clipping is applied in post transformation space. Pop() pops the clipping region pushed by the matching PushClip call:

```
20 void PushClip(Geometry clip);  
void Pop();.
```

Note that push operations can be arbitrarily nested as long as the pop operations are matched with a push. For

25 example, the following is valid:

```
PushTransform(...);  
DrawLine(...);  
PushClip(...);  
DrawLine(...);  
Pop();  
PushTransform(...);  
DrawRect(...);  
Pop();  
Pop();
```

A ProxyVisual is a visual that may be added more than once into the scene graph, e.g., below a container visual. Since any visual referred to by a ProxyVisual may be reached by multiple paths from the root, read services (TransformToDescendent, TransformFromDescendent and HitTest) do not work through a ProxyVisual. In essence, there is one canonical path from any visual to the root of the visual tree and that path does not include any ProxyVisuals.

FIG. 4 shows an example scene graph 400 in which ContainerVisuals and DrawingVisuals (and others) are related in a scene graph, and have associated data in the form of Instruction Lists, (e.g., in corresponding drawing contexts). The ContainerVisual is a container for Visuals, and ContainerVisuals can be nested into each other. The children of a ContainerVisual can be manipulated can be manipulated with via methods on the IVisual interface that the ContainerVisual implements. The order of the Visuals in the VisualCollection determines in which order the Visuals are rendered, i.e. Visuals are rendered from the lowest index to the highest index from back to front (painting order).

As described above, visuals can be drawn on by populating their drawing contexts with various drawing primitives, including Geometry, ImageSource and MediaData. Furthermore, there are a set of resources and classes that are shared
5 through this entire stack. This includes Pens, Brushes, Geometry, Transforms and Effects. The DrawingContext abstract class exposes a set of drawing operations that can be used to populate a DrawingVisual, ValidationVisual, ImageData, etc. In other words, the drawing context abstract class exposes a set
10 of drawing operations; for each drawing operation there are two methods, one that takes constants as arguments, and one that takes animators as arguments.

Geometry is a type of class (FIG. 10) that defines a vector graphics skeleton, without stroke or fill. Each
15 geometry object is a simple shape (LineGeometry, EllipseGeometry, RectangleGeometry), a complex single shape (PathGeometry) or a list of such shapes GeometryCollection with a combine operation (e.g., union, intersection, and so forth) specified. These objects form a class hierarchy as represented
20 in FIG. 10.

As represented in FIG. 11, the PathGeometry is a collection of Figure objects. In turn, each of the Figure objects is composed of one or more Segment objects which actually define the figure's shape. A Figure is a sub-section
25 of a Geometry that defines a segment collection. This segment

collection is a single connected series of two-dimensional Segment objects. The Figure can be either a closed shape with a defined area, or just a connected series of Segments that define a curve, but no enclosed area.

5 As represented in FIG. 12, when geometry (e.g., a rectangle) is drawn, a brush or pen can be specified, as described below. Furthermore, the pen object also has a brush object. A brush object defines how to graphically fill a plane, and there is a class hierarchy of brush objects. This
10 is represented in FIG. 12 by the filled rectangle 1202 that results when the visual including the rectangle and brush instructions and parameters is processed. A Pen object holds onto a Brush along with properties for Width, LineJoin, LineCap, MiterLimit, DashArray and DashOffset, as described
15 below. As also described below, some types of Brushes (such as gradients and nine grids) size themselves. When used, the size for these brushes is obtained from the bounding box, e.g., when the GradientUnits/DestinationUnits for the Brush is set to RelativeToBoundingBox, the bounding box of the primitive that
20 is being drawn is used. If those properties are set to Absolute, then the coordinate space is used.

The graphics object model of the present invention includes a Brush object model, which is generally directed towards the concept of covering a plane with pixels. Examples
25 of types of brushes are represented in the hierarchy of FIG.

13, and, under a Brush base class, include Gradient Brush, NineGridBrush, SolidColorBrush and TileBrush. GradientBrush includes LinearGradient and RadialGradient objects.

DrawingBrush and ImageBrush derive from TileBrush. Alternative
5 arrangements of the classes are feasible, e.g., deriving from TileBrush may be ImageBrush, VisualBrush, VideoBrush, NineGridBrush and Drawing Brush. Note that Brush objects may recognize how they relate to the coordinate system when they are used, and/or how they relate to the bounding box of the
10 shape on which they are used. In general, information such as size may be inferred from the object on which the brush is drawn. More particularly, many of the brush types use a coordinate system for specifying some of their parameters. This coordinate system can either be defined as relative to the
15 simple bounding box of the shape to which the brush is applied, or it can be relative to the coordinate space that is active at the time that the brush is used. These are known, respectively, as RelativeToBoundingBox mode and Absolute mode.

A SolidColorBrush object fills the identified plane with a
20 solid color. If there is an alpha component of the color, it is combined in a multiplicative way with the corresponding opacity attribute in the Brush base class. The following sets forth an example SolidColorBrush object:

```
25 public sealed class System.Windows.Media.SolidColorBrush : Brush  
    {
```

```

// Constructors
public SolidColorBrush(); // initialize to black
public SolidColorBrush(Color color);
5 public SolidColorBrush(System.Windows.Media.Animation.ColorComposer
   colorComposer);

// Properties
public Color Color { get; }
10 public IEnumerator ColorAnimations { get; }
}

public class System.Windows.Media.SolidColorBrushBuilder : BrushBuilder
{
15 // Constructors
public SolidColorBrushBuilder();
public SolidColorBrushBuilder(Color color);
public SolidColorBrushBuilder(SolidColorBrush scp);

// Properties
20 public Color Color { get; set; }
public AnimationList ColorAnimations { get; }

// Methods
25 public virtual Brush ToBrush();
}

```

The GradientBrush objects, or simply gradients, provide a gradient fill, and are drawn by specifying a set of gradient stops, which specify the colors along some sort of progression.

30 The gradient is by drawn by performing linear interpolations between the gradient stops in a gamma 2.2 RGB color space; interpolation through other gammas or other color spaces (HSB, CMYK and so forth, is also a feasible alternative. Two types of gradient objects include linear and radial gradients.

35 In general, gradients are composed of a list of gradient stops. Each of these gradient stops contains a color (with the included alpha value) and an offset. If there are no gradient stops specified, the brush is drawn as a solid transparent black, as if there were no brush specified at all. If there is
40 only one gradient stop specified, the brush is drawn as a solid

color with the one color specified. Like other resource classes, the gradient stop class (example in the table below) is derives from the changeable class and thus is selectively mutable, as described in the United States Patent Application
5 entitled "Changeable Class and Pattern to Provide Selective Mutability in Computer Programming Environments."

Gradients are drawn by specifying a set of gradient stops. These gradient stops specify the colors along some sort of progression. There are two types of gradients presently
10 supported, namely linear and radial gradients. The gradient is drawn by doing interpolations between the gradient stops in the specified color space.

Gradients are composed of a list of gradient stops. Each of these gradient stops contains a color (with the included
15 alpha value) and an offset. If there are no gradient stops specified, the brush is drawn as transparent (as if there were no brush specified). If there is only one gradient stop specified, the brush is drawn as a solid color with the one color specified. Any gradient stops with offsets in the range
20 of zero to one (0.0...1.0) are considered, with the largest stop in the range $(-\infty...0.0]$ and the smallest stop in the range $[1.0...+\infty)$. If the set of stops being considered includes a stop which is outside of the range zero to one, an implicit stop is derived at zero (and/or one) which represents the
25 interpolated color which would occur at this stop. Also, if

two or more stops are set at the same offset, a hard transition (rather than interpolated) occurs at that offset. The order in which stops are added determines the behavior at this offset; the first stop to be added is the effective color before that offset, the last stop to be set is the effective color after this stop, and any additional stops at this offset are ignored.

This class is a Changeable like other resource classes:

```
10 public sealed class System.Windows.Media.GradientStop : Changeable
    {
        public GradientStop(); public GradientStop(Color color, double offset);
        public GradientStop(Color color, ColorAnimationCollection colorAnimations,
            double offset, DoubleAnimationCollection offsetAnimations);
        public new GradientStop Copy(); // hides Changeable.Copy()
15
        // Default is transparent
        [Animation("ColorAnimations")]
        public Color Color { get; set; }
        public ColorAnimationCollection ColorAnimations { get; set; }
20
        // Default is 0
        [Animation("OffsetAnimations")]
        public double Offset { get; set; }
        public DoubleAnimationCollection OffsetAnimations { get; set; }
25 }
}
```

Like SolidColorBrush, this has nested Changeables in the animation collections.

The GradientSpreadMethod enum specifies how the gradient should be drawn outside of the specified vector or space. There are three possible values, including Pad, in which the end colors (first and last) are used to fill the remaining space, Reflect, in which the stops are replayed in reverse

order repeatedly to fill the space, and Repeat, in which the stops are repeated in order until the space is filled. The default value for properties of this type is Pad:

```
5 public enum System.Windows.Media.GradientSpreadMethod
  {
    Pad,
    Reflect,
    Repeat
  }
10
```

FIGS. 14 and 15 provide some GradientSpreadMethod examples, (albeit in grayscale rather than in color). Each shape has a linear gradient going from white to grey. The solid line represents the gradient vector.

15 In general, a LinearGradientBrush is used to fill an area with a linear gradient. A linear gradient defines a gradient along a line. The line's end point is defined by the linear gradient's StartPoint and EndPoint properties. By default, the StartPoint of a linear gradient is (0,0), the upper-left corner
20 of the area being filled, and its EndPoint is (1,1), the bottom-right corner of the area being filled. As represented in FIG. 15, using the default values, the colors in the resulting gradient are interpolated along a diagonal path. The black line formed from the start and end points of the gradient
25 has been added herein to highlight the gradient's interpolation path.

The `ColorInterpolationMode` enum defines the interpolation mode for colors within a gradient. The two options are `PhysicallyLinearGamma10` and `PerceptuallyLinearGamma22`.

```
5 public enum ColorInterpolationMode
  {
    // Colors are interpolated in Gamma 1.0 space
    PhysicallyLinearGamma10,
10    // Colors are interpolated in Gamma 2.2 space
    PerceptuallyLinearGamma22
  }
```

This is an abstract base class.

```
15 public abstract class System.Windows.Media.GradientBrush : Brush
  {
    internal GradientBrush();
    public new GradientBrush Copy(); // hides Changeable.Copy()

20    // Default is "PerceptuallyLinearGamma22"
    public ColorInterpolationMode ColorInterpolationMode { get; set; }

    // Default is RelativeToBoundingBox
25    public BrushMappingMode MappingMode { get; set; }

    // Default is Pad
    public GradientSpreadMethod SpreadMethod { get; set; }

30    // Gradient Stops
    public void AddStop(Color color, double offset);
    public GradientStopCollection GradientStops { get; set; }

    // ColorInterpolationMode
35    public ColorInterpolationMode ColorInterpolationMode { get; set; }
  }
```

As described above in the Changeables section, GradientBrush is a complex-type with respect to Changeables, because its GradientStops property itself holds Changeables. That means that GradientBrush needs to implement the protected methods MakeUnchangeableCore(), and PropagateEventHandler(), as well as CloneCore() that Changeable subclasses implement. It may also choose to implement ValidateObjectState() if there are invalid combinations of GradientStops that make up the collection, for instance.

10 The LinearGradient specifies a linear gradient brush along a vector. The individual stops specify colors stops along that vector.

```
public sealed class System.Windows.Media.LinearGradient :
GradientBrush
{
    public LinearGradient(); // initializes to transparent

    // Sets up a gradient with two colors and a gradient
vector
    // specified to fill the object the gradient is applied
to.
    // This implies RelativeToBoundingBox for the
GradientUnits
    // property
    public LinearGradient(Color color1, Color color2, double
angle);
    public LinearGradient(Color color1, Color color2,
        Point vectorStart, Point vectorEnd);
    public new LinearGradient Copy(); // hides
Changeable.Copy()

    // Gradient Vector Start Point
    // Default is 0,0
    [Animation("StartPointAnimations")]
    public Point StartPoint { get; set; }
    public PointAnimationCollection StartPointAnimations {
get; set; }
```

```

        // Default is 1,1
        [Animation("EndPointAnimations")]
        public Point EndPoint { get; set; }
        public PointAnimationCollection EndPointAnimations { get;
set; }
    }

    linear-gradient-brush:
        "HorizontalGradient" comma-wsp color comma-wsp color |
        "VerticalGradient" comma-wsp color comma-wsp color |
        "LinearGradient" comma-wsp coordinate-pair comma-wsp
color comma-wsp color

```

The markup for LinearGradient allows specification of a LinearGradient with two color stops, at offsets zero and one. If the "LinearGradient" version is used, the start point and end point are specified, respectively. If "HorizontalGradient" is used, the start point is set to 0,0 and the end point is set to 1,0. If "VerticalGradient" is used, the start point is set to 0,0 and the end point is set to 0,1. In these cases, the default MappingMode is used, which is RelativeToBoundingBox.

The RadialGradient is similar in programming model to the linear gradient. However, whereas the linear gradient has a start and end point to define the gradient vector, the radial gradient has a circle along with a focal point to define the gradient behavior. The circle defines the end point of the gradient - in other words, a gradient stop at 1.0 defines the color at the circle's circumference. The focal point defines center of the gradient. A gradient stop at 0.0 defines the color at the focal point. FIG. 16 represents a RadialGradient

that (in grayscale) goes from white to grey. The outside circle represents the gradient circle while the solid dot denotes the focal point. This gradient has SpreadMethod set to Pad.

```
public sealed class System.Windows.Media.RadialGradient :
GradientBrush
{
    public RadialGradient(); // initialize to transparent

    // Sets up a gradient with two colors.
    // This implies RelativeToBoundingBox for the
GradientUnits
    // property along with a center at (0.5,0.5)
    // a radius of 0.5 and a focal point at (0.5,0.5)
    public RadialGradient(Color color1, Color color2);

    public new RadialGradient Copy(); // hides
Changeable.Copy()

    // Default is 0.5,0.5
    [Animation("CenterAnimations")]
    public Point Center { get; set; }
    public PointAnimationCollection CenterAnimations { get;
set; }

    // Default is 0.5
    [Animation("RadiusXAnimations")]
    public double RadiusX { get; set; }
    public DoubleAnimationCollection RadiusXAnimations { get;
set; }

    // Default is 0.5
    [Animation("RadiusYAnimations")]
    public double RadiusY { get; set; }
    public DoubleAnimationCollection RadiusYAnimations { get;
set; }

    // Default is 0.5,0.5
    [Animation("FocusAnimations")]
    public Point Focus { get; set; }
    public PointAnimationCollection FocusAnimations { get;
set; }
}
```

The markup for RadialGradient allows specification of a RadialGradient with two color stops, at offsets 0 and 1 respectively. The default MappingMode is used, which is RelativeToBoundingBox, as are the default radii, 0.5:

5

```
radial-gradient-brush:
  "RadialGradient" comma-wsp color comma-wsp color
```

The TileBrush is an abstract base class which contains logic to describe a tile and a means by which that tile should fill an area. Subclasses of TileBrush contain content, and logically define a way to fill an infinite plane.

The Stretch enum is used to describe how a ViewBox (source coordinate space) is mapped to a ViewPort (destination coordinate space). This is used in TileBrush:

```
public enum System.Windows.Stretch
{
    // Preserve original size
    None,
    // Aspect ratio is not preserved, ViewBox fills ViewPort
    Fill,
    // Aspect ratio is preserved, ViewBox is uniformly scaled
    // as large as possible such that both width and height fit within
    // ViewPort
    Uniform,
    // Aspect ratio is preserved, ViewBox is uniformly scaled
    // as small as possible such that the entire ViewPort is filled by
    // the ViewBox
    UniformToFill
}
```

15

FIG. 18 provides stretch examples. In these examples, the contents are top/left aligned.

The TileMode enum is used to describe if and how a space is filled by Tiles. A TileBrush defines where the base Tile is (specified by the ViewPort). The rest of the space is filled based on the TileMode value.

```
public enum System.Windows.Media.TileMode
{
    // Do not tile - only the base tile is drawn, the remaining
    // area is
    // left as transparent
    None,
    // The basic tile mode - the base tile is drawn and the
    // remaining area
    // is filled by repeating the base tile such that the right
    // edge of one
    // tile butts the left edge of the next, and similarly for
    // bottom and top
    Tile,
    // The same as tile, but alternate columns of tiles are
    // flipped
    // horizontally. The base tile is drawn untransformed.
    FlipX,
    // The same as tile, but alternate rows of tiles are
    // flipped vertically
    // The base tile is drawn untransformed.
    FlipY,
    // The combination of FlipX and FlipY. The base tile is
    // drawn
    // untransformed
    FlipXY
}
```

FIG. 19 provides TileMode examples. The top left-most tile in each example is the base tile. These examples represent None, Tile, FlipX, FlipY and FlipXY.

The VerticalAlignment enum is used to describe how content is positioned within a container vertically:

```

public enum System.Windows.VerticalAlignment
{
    // Align contents towards the top of a space
    Top,
    // Center contents vertically
    Center,
    // Align contents towards the bottom of a space
    Bottom,
}

```

The HorizontalAlignment enum is used to describe how content is positioned within a container horizontally.

```

public enum System.Windows.HorizontalAlignment
{
    // Align contents towards the left of a space
    Left,
    // Center contents horizontally
    Center,
    // Align contents towards the right of a space
    Right,
}

```

- 5 The TileBrush properties select a rectangular portion of the infinite plane to be a tile (the ViewBox) and describe a destination rectangle (ViewPort) which will be the base Tile in the area being filled. The remaining destination area will be filled based on the TileMode property, which controls if and
- 10 how the original tile is replicated to fill the remaining space:

```

public abstract class System.Windows.Media.TileBrush : Brush
{
    public new TileBrush Copy(); // hides Brush.Copy()

    // Default is RelativeToBoundingBox
    public BrushMappingMode ViewPortUnits { get; set; }

    // Default is RelativeToBoundingBox
    public BrushMappingMode ContentUnits { get; set; }
}

```

```

        // Default is Rect.Empty
        [Animation("ViewBoxAnimations")]
        public Rect ViewBox { get; set; }
        public RectAnimationCollection ViewBoxAnimations { get;
set; }

        // Default is Fill
        public Stretch Stretch { get; set; }

        // Default is None
        public TileMode TileMode { get; set; }

        // Default is Center
        public HorizontalAlignment HorizontalAlignment { get; set;
}

        // Default is Center
        public VerticalAlignment VerticalAlignment { get; set; }

        // Default is 0,0,1,1
        [Animation("ViewportAnimations")]
        public Rect ViewPort { get; set; }
        public RectAnimationCollection ViewPortAnimations { get;
set; }
}

```

A TileBrush's contents have no intrinsic bounds, and effectively describe an infinite plane. These contents exist in their own coordinate space, and the space which is being

5 filled by the TileBrush is the local coordinate space at the time of application. The content space is mapped into the local space based on the ViewBox, ViewPort, Alignments and Stretch properties. The ViewBox is specified in content space, and this rectangle is mapped into the ViewPort rectangle.

10 The ViewPort defines the location where the contents will eventually be drawn, creating the base tile for this Brush. If the value of ViewPortUnits is Absolute, the value of ViewPort

is considered to be in local space at the time of application.
If, instead, the value of ViewPortUnits is
RelativeToBoundingBox, then the value of ViewPort is considered
to be in the coordinate space where 0,0 is the top/left corner
5 of the bounding box of the object being painted and 1,1 is the
bottom/right corner of the same box. For example, consider a
RectangleGeometry being filled which is drawn from 100,100 to
200,200. Then, if the ViewPortUnits is Absolute, a ViewPort of
(100,100,100,100) would describe the entire content area. If
10 the ViewPortUnits is RelativeToBoundingBox, a ViewPort of
(0,0,1,1) would describe the entire content area. If the
ViewPort's Size is empty and the Stretch is not None, this
Brush renders nothing.

The ViewBox is specified in content space. This rectangle
15 is transformed to fit within the ViewPort as determined by the
Alignment properties and the Stretch property. If the Stretch
is None, then no scaling is applied to the contents. If the
Stretch is Fill, then the ViewBox is scaled independently in
both X and Y to be the same size as the ViewPort. If the
20 Stretch is Uniform or UniformToFill, the logic is similar but
the X and Y dimensions are scaled uniformly, preserving the
aspect ratio of the contents. If the Stretch is Uniform, the
ViewBox is scaled to have the more constrained dimension equal
to the ViewPort's size. If the Stretch is UniformToFill, the
25 ViewBox is scaled to have the less constrained dimension equal

to the ViewPort's size. Another way to think of this is that both Uniform and UniformToFill preserve aspect ratio, but Uniform ensures that the entire ViewBox is within the ViewPort (potentially leaving portions of the ViewPort uncovered by the ViewBox), and UniformToFill ensures that the entire ViewPort is filled by the ViewBox (potentially causing portions of the ViewBox to be outside the ViewPort). If the ViewBox's area is empty, then no Stretch will apply. Alignment will still occur, and it will position the "point" ViewBox.

Once the ViewPort is determined (based on ViewPortUnits) and the ViewBox's destination size is determined (based on Stretch), the ViewBox needs to be positioned within the ViewPort. If the ViewBox is the same size as the ViewPort (if Stretch is Fill, or if it just happens to occur with one of the other three Stretch values), then the ViewBox is positioned at the Origin so as to be identical to the ViewPort. If not, then HorizontalAlignment and VerticalAlignment are considered. Based on these properties, the ViewBox is aligned in both X and Y dimensions. If the HorizontalAlignment is Left, then the left edge of the ViewBox will be positioned at the Left edge of the ViewPort. If it is Center, then the center of the ViewBox will be positioned at the center of the ViewPort, and if Right, then the right edges will meet. The process is repeated for the Y dimension.

If the ViewBox is Empty it is considered unset. If it is unset, then ContentUnits are considered. If the ContentUnits are Absolute, no scaling or offset occurs, and the contents are drawn into the ViewPort with no transform. If the ContentUnits
5 are RelativeToBoundingBox, then the content origin is aligned with the ViewPort Origin, and the contents are scaled by the object's bounding box's width and height.

When filling a space with a TileBrush, the contents are mapped into the ViewPort as above, and clipped to the ViewPort.
10 This forms the base tile for the fill, and the remainder of the space is filled based on the Brush's TileMode. If set, the Brush's transform is applied, which occurs after the other mapping, scaling, offsetting, and so forth.

A VisualBrush is a TileBrush whose contents are specified
15 by a Visual. This Brush can be used to create complex patterns, or it can be used to draw additional copies of the contents of other parts of the scene.

```
public sealed class System.Windows.Media.VisualBrush :  
TileBrush  
{  
    public VisualBrush(); // initializes to transparent  
    public VisualBrush(Visual v);  
  
    public new VisualBrush Copy(); // hides TileBrush.Copy()  
  
    // Visual - Default is null (transparent Brush)  
    public Visual Visual { get; set; }  
}
```

ImageBrush is a TileBrush having contents specified by an ImageData. This Brush can be used to fill a space with an Image.

```
public sealed class System.Windows.Media.ImageBrush :
TileBrush
{
    public ImageBrush(); // Initializes to transparent
contents

    // Sets the image, sets ViewBox to (0,0,Width,Height)
// and Stretch to Fill
    public ImageBrush(ImageData image);

    public new ImageBrush Copy(); // hides TileBrush.Copy()

    // Default is null
    public ImageData ImageData { get; set; }

    // Default is true
    // If this is true, the ViewBox property will be
overridden
    // and effectively set to the native size of the Image
    public bool SizeViewBoxToContent { get; set; }
}
```

5 VideoBrush is a TileBrush having contents specified by a VideoData. This Brush can be used to fill a space with a Video.

```
public sealed class System.Windows.Media.VideoBrush :
TileBrush
{
    public VideoBrush(); // Initializes to transparent
contents

    // Sets the image, sets ViewBox to (0,0,Width,Height) and
the
    // Stretch to Fill
    public VideoBrush(VideoData video);

    public new VideoBrush Copy(); // hides TileBrush.Copy()

    // Default is null
    public VideoData VideoData { get; set; }
}
```

```
    // Default is true
    // If this is true, the VBox property will be
    overridden
    // and effectively set to the native size of the Video
    public bool SizeVBoxToContent { get; set; }
}
```

NineGridBrush is a Brush which always fills the object bounding box with its content image, and the image stretch isn't accomplished purely via a visual scale. The Image source
5 is divided into nine rectangles by four borders (hence the name NineGrid). The contents of the image in each of those nine regions are scaled in 0, 1 or 2 dimensions until they fill the object bounding box. The dimensions in which each section is scaled can be seen in this diagram: FIG. 17 represents the
10 concept of a NineGrid, being enlarged from a first instance 1702 to a second instance 1704, with four types of showing the nine grids which are defined by the Top, Left, Bottom and Right borders. The arrows in each grid square show the dimension(s) in which those contents will be stretched to meet the ViewPort
15 size.

In addition to the nine grid regions pictured above, there is an optional "tenth" grid. This takes the form of an additional image which is centered in the ViewPort and which is not scaled. This can be used to place a shape in the center of
20 a button, etc. This "tenth grid" is called a glyph, and is exposed by the GlyphImageData property:

```

public sealed class System.Windows.Media.NineGridBrush : Brush
{
    public NineGridBrush(ImageData imageData,
                        int LeftBorder,
                        int RightBorder,
                        int TopBorder,
                        int BottomBorder);

    public NineGridBrush(ImageData imageData,
                        int LeftBorder,
                        int RightBorder,
                        int TopBorder,
                        int BottomBorder,
                        ImageData glyphImage);

    public new NineGridBrush Copy(); // hides Brush.Copy()

    // Default is null
    public ImageData ImageData { get; set; }

    // Default is 0
    public int LeftBorder { get; set; }

    // Default is 0
    public int RightBorder { get; set; }

    // Default is 0
    public int TopBorder { get; set; }

    // Default is 0
    public int BottomBorder { get; set; }

    // Default is null
    public ImageData GlyphImageData { get; set; }
}

```

Note that the border members count in from the edge of the image in image pixels

The Pen is an object that takes a Brush and other
5 parameters that describe how to stroke a space/Geometry.
Conceptually, a Pen describes how to create a stroke area from
a Geometry. A new region is created which is based on the
edges of the Geometry, the Pen's Thickness, the PenLineJoin,

PenLineCap, and so forth. Once this region is created, it is filled with the Brush.

```
public sealed class System.Windows.Media.Pen : Changeable
{
    // Constructors
    Public Pen();
    public Pen(Brush brush, double thickness);

    public new Pen Copy(); // hides Changeable.Copy()

    // Properties
    // Default is DashArrays.Solid (no dashes)
    public DoubleCollection DashArray { get; set;}

    // Default is 0
    [Animations(DashOffsetAnimations)]
    public double DashOffset { get; set;}
    public DoubleAnimationCollection DashOffsetAnimations {
get; set;}

    // Default is Flat
    public PenLineCap StartLineCap { get; set;}

    // Default is Flat
    public PenLineCap EndLineCap { get; set;}

    // Default is Flat
    public PenDashCap DashCap { get; set;}

    // Default is Miter
    public PenLineJoin LineJoin { get; set;}

    // Default is 10
    public double MiterLimit { get; set;}

    // Default is null
    public Brush Brush { get; set;}

    // Default is 1.0
    [Animations(ThicknessAnimations)]
    public double Thickness { get; set;}
    public DoubleAnimationCollection ThicknessAnimations {
get; set;}
}
```

The PenLineCap determines how the ends of a stroked line are drawn:

```
public enum System.Windows.Media.PenLineCap
{
    // This is effectively no line cap - the line is squared
off    off
    // at the last point in the line
    Flat,
    // The line is capped by a hemi-circle of diameter equal
to    to
    // the line width
    Round,
    // The dash is capped by a triangle
    Triangle,
    // The line is capped with a square of side with equal to
the    the
    // line width, centered on the end point
    Square
}
```

- 5 The PenDashCap determines how the ends of each dash in a dashed, stroked line are drawn:

```
public enum System.Windows.Media.PenDashCap
{
    // This is effectively no dash cap - the line is squared
off    off
    // at the last point in the line
    Flat,
    // The dash is capped by a hemi-circle of diameter equal
to    to
    // the line width
    Round,
    // The dash is capped by a triangle
    Triangle
}
```

The PenLineJoin determines how joints are draw when stroking a line:

```
public enum System.Windows.Media.PenLineJoin
{
```

```

    // A sharp corner is created at the intersection of the
    outer
    // edges of the intersecting line segments
    Miter,
    // Similar to Miter, but the corner is rounded
    Round,
    // A beveled join, this produces a diagonal corner
    Bevel
}

```

The DashArrays class comprises static properties which provide access to common, well-known dash styles:

```

public sealed System.Windows.Media.DashArrays
{
    // A solid Dash array (no dashes)
    public static DoubleCollection Solid { get; }
    // Dash - 3 on, 1 off
    public static DoubleCollection Dash { get; }
    // Dot - 1 on, 1 off
    public static DoubleCollection Dot { get; }
    // DashDot - 3 on, 1 off, 1 on, 1 off
    public static DoubleCollection DashDot { get; }
    // DashDotDot - 3 on, 1 off, 1 on, 1 off, 1 on, 1 off
    public static DoubleCollection DashDotDot { get; }
}

```

5

Another brush object represented in FIG. 13 is a VisualBrush object. A VisualBrush is a TileBrush whose contents are specified by a Visual. This Brush can be used to create complex patterns, or it can be used to draw additional copies of the contents of other parts of the scene.

```

public sealed class System.Windows.Media.VisualBrush :
TileBrush
{
    public VisualBrush(); // initializes to transparent
    public VisualBrush(Visual v);

    public new VisualBrush Copy(); // hides TileBrush.Copy()

    // Visual - Default is null (transparent Brush)
    public Visual Visual { get; set; }
}

```

Conceptually, the VisualBrush provides a way to have a visual drawn in a repeated, tiled fashion as a fill. This is represented in FIG. 12 by the visual brush referencing a visual (and any child visuals) that specifies a single circular shape 1220, with that circular shape filling a rectangle 1222. Thus, the VisualBrush object may reference a visual to define how that brush is to be drawn, which introduces a type of multiple use for visuals. In this manner, a program may use an arbitrary graphics "metafile" to fill an area via a brush or pen. Since this is a compressed form for storing and using arbitrary graphics, it serves a graphics resource.

In one implementation, a VisualBrush's contents have no intrinsic bounds, and effectively describe an infinite plane. These contents exist in their own coordinate space, and the space which is being filled by the VisualBrush is the local coordinate space at the time of application. The content space is mapped into the local space based on the ViewBox, ViewPort, Alignments and Stretch properties. The ViewBox is specified in

content space, and this rectangle is mapped into the ViewPort (as specified via the Origin and Size properties) rectangle.

The ViewPort defines the location where the contents will eventually be drawn, creating the base tile for this Brush. If
5 the value of DestinationUnits is UserSpaceOnUse, the Origin and Size properties are considered to be in local space at the time of application. If instead the value of DestinationUnits is ObjectBoundingBox, then an Origin and Size are considered to be in the coordinate space, where 0,0 is the top/left corner of
10 the bounding box of the object being brushed, and 1,1 is the bottom/right corner of the same box. For example, consider a RectangleGeometry being filled which is drawn from 100,100 to 200,200. In such an example, if the DestinationUnits is UserSpaceOnUse, an Origin of 100,100 and a Size of 100,100
15 would describe the entire content area. If the DestinationUnits is ObjectBoundingBox, an Origin of 0,0 and a Size of 1,1 would describe the entire content area. If the Size is empty, this Brush renders nothing.

The ViewBox is specified in content space. This rectangle
20 is transformed to fit within the ViewPort as determined by the Alignment properties and the Stretch property. If the Stretch is none, then no scaling is applied to the contents. If the Stretch is Fill, then the ViewBox is scaled independently in both X and Y to be the same size as the ViewPort. If the
25 Stretch is Uniform or UniformToFill, the logic is similar but

the X and Y dimensions are scaled uniformly, preserving the aspect ratio of the contents. If the Stretch is Uniform, the ViewBox is scaled to have the more constrained dimension equal to the ViewPort's size. If the Stretch is UniformToFill, the
5 ViewBox is scaled to have the less constrained dimension equal to the ViewPort's size. In other words, both Uniform and UniformToFill preserve aspect ratio, but Uniform ensures that the entire ViewBox is within the ViewPort (potentially leaving portions of the ViewPort uncovered by the ViewBox), and
10 UniformToFill ensures that the entire ViewPort is filled by the ViewBox (potentially causing portions of the ViewBox to be outside the ViewPort). If the ViewBox is empty, then no Stretch will apply. Note that alignment will still occur, and it will position the "point" ViewBox.

15 FIG. 18 provides representations of a single tile 1800 of graphics rendered with various stretch settings, including a tile 1800 when stretch is set to "none." The tile 1802 is a representation of when the stretch is set to "Uniform," the tile 1804 when stretch is set to "UniformToFill," and the tile
20 1806 when stretch is set to "Fill."

Once the ViewPort is determined (based on DestinationUnits) and the ViewBox's size is determined (based on Stretch), the ViewBox needs to be positioned within the ViewPort. If the ViewBox is the same size as the ViewPort (if
25 Stretch is Fill, or if it just happens to occur with one of the

other three Stretch values), then the ViewBox is positioned at the Origin so as to be identical to the ViewPort. Otherwise, HorizontalAlignment and VerticalAlignment are considered.

Based on these properties, the ViewBox is aligned in both X and
5 Y dimensions. If the HorizontalAlignment is Left, then the left edge of the ViewBox will be positioned at the Left edge of the ViewPort. If it is Center, then the center of the ViewBox will be positioned at the center of the ViewPort, and if Right, then the right edges will meet. The process is repeated for
10 the Y dimension.

If the ViewBox is (0,0,0,0), it is considered unset, whereby ContentUnits are considered. If the ContentUnits are UserSpaceOnUse, no scaling or offset occurs, and the contents are drawn into the ViewPort with no transform. If the
15 ContentUnits are ObjectBoundingBox, then the content origin is aligned with the ViewPort Origin, and the contents are scale by the object's bounding box's width and height.

When filling a space with a VisualBrush, the contents are mapped into the ViewPort as above, and clipped to the ViewPort.
20 This forms the base tile for the fill, and the remainder of the space is filled based on the Brush's TileMode. Finally, if set, the Brush's transform is applied - it occurs after all the other mapping, scaling, offsetting, etc.

The TileMode enumeration is used to describe if and how a
25 space is filled by its Brush. A Brush which can be tiled has a

tile rectangle defined, and this tile has a base location within the space being filled. The rest of the space is filled based on the TileMode value. FIG. 19 provides a representation of example graphics with various TileMode settings, including "None" 1900, "Tile" 1092, "FlipX" 1904, "FlipY" 1906 and "FlipXY" 1908. The top left-most tile in the various example graphics comprises the base tile.

FIG. 20 represents a VisualBrush Grid that is defined for the tiles in a VisualBrush. The first circle is a simple grid, and the second has a Transform with a Skew in the x direction of 47. FIG. 21 shows this being filled with an image.

Returning to FIG. 13, image brush derives from tile brush and thus can be tiled. NineGridBrush is very similar to ImageBrush except the image is warped based on the size. In essence, NineGridBrush may be thought of a custom type of Stretch, in which certain parts of the image stretch, while others (e.g., borders) do not. Thus, while the Size of the image in the ImageBrush will cause a simple scale, the NineGridBrush will produce a non-uniform scale up to the desired size. The units for the non-scaled areas are the user units when the brush is applied, which means that ContentUnits (if it existed for NineGridBrush) would be set to UserUnitsOnUse. The Transform property of the Brush can be used effectively. Note that the border members count in from the edge of the image.

As generally described above, the graphics object model of the present invention includes a Transform object model, which includes the types of transforms represented in the hierarchy of FIG. 7, under a Transform base class. These different types of components that make up a transform may include TransformList, TranslateTransform, RotateTransform, ScaleTransform, SkewTransform, and MatrixTransform. Individual properties can be animated, e.g., a program developer can animate the Angle property of a RotateTransform.

Matrices for 2D computations are represented as a 3x3 matrix. For the needed transforms, only six values are needed instead of a full 3x3 matrix. These are named and defined as follows.

$$\begin{bmatrix} m00 & m01 & 0 \\ m10 & m11 & 0 \\ m20 & m21 & 1 \end{bmatrix}$$

When a matrix is multiplied with a point, it transforms that point from the new coordinate system to the previous coordinate system:

$$\begin{bmatrix} X_{\text{newCoordSys}} & y_{\text{newCoordSys}} & 1 \end{bmatrix} \bullet \begin{bmatrix} m00 & m01 & 0 \\ m10 & m11 & 0 \\ m20 & m21 & 1 \end{bmatrix} = \begin{bmatrix} X_{\text{oldCoordSys}} & y_{\text{oldCoordSys}} & 1 \end{bmatrix}$$

Transforms can be nested to any level. Whenever a new transform is applied it is the same as post-multiplying it onto the current transform matrix:

5

$$\begin{bmatrix} X_{\text{newCoordSys}} & Y_{\text{newCoordSys}} & 1 \end{bmatrix} \cdot \begin{bmatrix} m00_2 & m01_2 & 0 \\ m10_2 & m11_2 & 0 \\ m20_2 & m21_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} m00_1 & m01_1 & 0 \\ m10_1 & m11_1 & 0 \\ m20_1 & m21_1 & 1 \end{bmatrix} = \begin{bmatrix} X_{\text{oldCoordSys}} & Y_{\text{oldCoordSys}} & 1 \end{bmatrix}$$

Most places in the API do not take a Matrix directly, but instead use the Transform class, which supports animation.

10

15

20

25

30

35

40

```
public struct System.Windows.Media.Matrix
{
    // Construction and setting
    public Matrix(); // defaults to identity

    public Matrix(
        double m00, double m01,
        double m10, double m11,
        double m20, double m21);

    // Identity
    public static readonly Matrix Identity;
    public void SetIdentity();
    public bool IsIdentity { get; }

    public static Matrix operator *(Matrix matrix1, Matrix matrix2);
    public static Point operator *(Matrix matrix, Point point);

    // These function reinitialize the current matrix with
    // the specified transform matrix.
    public void SetTranslation(double dx, double dy);
    public void SetTranslation(Size offset);
    public void SetRotation(double angle); // degrees
    public void SetRotation(double angle, Point center); // degrees
    public void SetRotationRadians(double angle);
    public void SetRotationRadians(double angle, Point center);
    public void SetScaling(double sx, double sy);
    public void SetScaling(double sx, double sy, Point center);
    public void SetSkewX(double angle); // degrees
    public void SetSkewY(double angle); // degrees
    public void SetSkewXRadians(double angle);
    public void SetSkewYRadians(double angle);

    // These function post-multiply the current matrix
    // with the specified transform
}
```

```

    public void ApplyTranslation(double dx, double dy);
    public void ApplyTranslation(Size offApply);
    public void ApplyRotation(double angle); // degrees
    5   public void ApplyRotation(double angle, Point center); //
        degrees
    public void ApplyRotationRadian(double angle);
    public void ApplyRotationRadian(double angle, Point center);
    public void ApplyScaling(double sx, double sy);
    10   public void ApplyScaling(double sx, double sy, Point center);
    public void ApplySkewX(double angle); // degrees
    public void ApplySkewY(double angle); // degrees
    public void ApplySkewXRadians(double angle);
    public void ApplySkewYRadians(double angle);
    15   public void ApplyMatrix(Matrix matrix);

    // Inversion stuff
    public double Determinant { get; }
    public bool IsInvertible { get; }
    20   public void Invert(); // Throws ArgumentException if
        !IsInvertible
    public static Matrix Invert(Matrix matrix);

    // Individual members
    public double M00 { get; set; }
    25   public double M01 { get; set; }
    public double M10 { get; set; }
    public double M11 { get; set; }
    public double M20 { get; set; }
    public double M21 { get; set; }
    30   };

```

CONCLUSION

As can be seen from the foregoing detailed description,
 35 there is provided a system, method and object model that
 provide program code with the ability to interface with a scene
 graph. The system, method and object model are straightforward
 to use, yet powerful, flexible and extensible.

While the invention is susceptible to various
 40 modifications and alternative constructions, certain
 illustrated embodiments thereof are shown in the drawings and

have been described above in detail. It should be understood, however, that there is no intention to limit the invention to the specific forms disclosed, but on the contrary, the intention is to cover all modifications, alternative
5 constructions, and equivalents falling within the spirit and scope of the invention.